

**EE 254**

# **Verilog Tutorial**

Dr. D. K. Blandford  
Department of Electrical Engineering and Computer Science  
University of Evansville  
February 23, 2012

Copyright © 2012

## Verilog Tutorial

### 1.0 Syntax – comments, punctuation, variable names, signal values, constants, parameters, and memory.

- Comments – Verilog comments are the same as in C++. Use // for a single line comment or /\* ... \*/ for a multiline comment.
- Punctuation – white spaces are ignored in Verilog. A semicolon is used to indicate the end of a command line and commas are typically used to separate elements in a list. Like C++, Verilog is case sensitive.
- Identifiers – An identifier is usually a variable. You can use any letter, digit, the underscore, or \$. Identifiers may not begin with a digit and may not be the same as a Verilog key word. As in C++ variable names should be chosen to assist in documentation.
- Signal values – signals in Verilog have one of four values. These are 0 (logic 0), 1 (logic 1), X, or x ( don't care or unknown), and Z or z for high impedance tri-state.
- Constants – The generic **declaration for a constant** in Verilog is

**[size] ['radix] constant\_value**

In this declaration size indicates the number of bits and 'radix gives the number base (d = decimal, b = binary, o = octal, h = hex). The default radix is decimal.

Examples

```
16           //The number 16 base 10
4'b1010     //The binary number 1010
8'bx        //An 8-bit binary number of unknown value
12'habc     //The hex number abc = 1010 1011 1100 in binary
8'b10       //The binary number 0000 0010
```

- Parameters – a parameter in Verilog can be any Verilog constant. Parameters are used to generalize a design. For example a 4-bit adder becomes more useful as a design if it is put together as an n-bit adder where n is a parameter specified by the user before compilation. Parameter declarations are done immediately after the module declaration. Here are some typical parameter examples:

```
parameter n = 12;
parameter [3:0]p1 = 4'b1011;
parameter n = 12, m = 32;
```

- Memory – Verilog allows for two dimensional arrays which typically get used for memory spaces. For example **reg[7:0] m[63:0];** declares m to be a two-dimensional array consisting of 64 eight-bit words. You can access any word as m[2] for example, but you do not get access to the bits in the word unless you copy the word to another 8-bit reg variable.<sup>1</sup>

### 2.0 Structure – Modules, ports, and signals.

- Module – A module in Verilog is used to define a circuit or a sub-circuit. The module is the fundamental circuit building block in Verilog. Modules have the following

<sup>1</sup> Verilog 2001 supports 2-level addressing such as m[2][3] so you can get at individual bits. Verilog 2001 also adds more than two dimensions for arrays.

structure: (keywords in bold). Note that the module declaration ends with a semicolon but the keyword **endmodule** does not.

```
module module_name (port_name list);
    [declarations]
    [assign statements]
    [initial block]
    [always block]
    [gate instantiations]
    [other module instantiations]
endmodule
```

- Ports – Ports in Verilog can be of type **input**, **output**, or **inout**. The module ports are given in the port name list and are declared in the beginning of the module. Here is a sample module with input and output ports.

```
module MyModule(aIn, bOut);
    input aIn;
    output bOut;
    ...
endmodule
```

The port names **input** and **output** default to type **wire**. Either can be a vector and the output variables can be of redeclared to type **reg**. The output and input variables in a module are typically names for the output and input pins on the implementation chip.

- Signals – a signal is represented by either a net type or a variable type in Verilog. The net type represents a circuit node and these can be of several types. The two net types most often used are **wire** and **tri**. Type nets do not have to be declared in Verilog since Verilog assumes that all signals are nets unless they are declared otherwise. Variables are either of type **reg** or **integer**. Integers are always 32-bits where the reg type of variables may be of any length. Typically we use integers as loop counters and reg variables for all other variables. The generic form for representing a signal in Verilog is:

```
type[range] signal_name
```

The range is omitted for scalar variables but used for vectors.

The net types are typically used for input signals and for intermediate signals within combinational logic. Variables are used for sequential circuits or for outputs which are assigned a value within a sequential always block.

Examples:

```
wire w;           //w is a single net of type wire
wire[2:0] wVect; //Declares wVect[2], wVect[1], wVect[0]
tri[7:0] bus      //An 8-bit tri state bus
integer i;        //i is a 32-bit integer used for loop control
reg r;            //r is a 1-bit register
reg[7:0] buf;     //buf is an 8-bit register
reg[3:0] r1, r2   //r1 and r2 are both 4-bit registers
```

### 3.0 Combinational Circuits – Gates, continuous assignment, and operators.

- Gate – The general form for declaring the instance of a gate in Verilog is

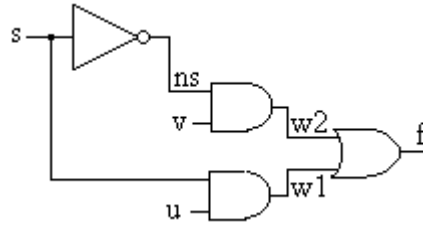
```
gate_type [gate_name](out_port, in_port ...);
```

The gate\_type specifies the type of gate you want to use such as **and**, **or**, **xor**, etc.

The gate name is optional and is user assigned. The port list in parenthesis, typically

consists of an output port (always first) followed by a comma separated list of input ports. Here are some gate instantiation examples:

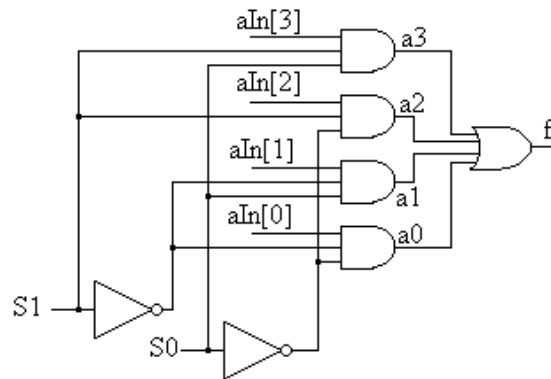
```
module AndOr(f, u, v, s);
  input s, u, v;
  output f;
  wire w1, w2;
  wire ns;
  and A1(w1, u, s);
  not N1(ns, s);
  and A2(w2, v, ns);
  or O1(f, w1, w2);
endmodule
```



**Figure 1**

Verilog code for a 2 to 1 multiplexer

```
module Mux4To1(f, s0, s1, aIn);
  output f;
  input s0, s1;
  input [3:0]aIn;
  wire ns0, ns1;
  wire a0, a1, a2, a3;
  //
  not nots0(ns0, s0);
  not nots1(ns1, s1);
  //
  and and0(a0, ns0, ns1, aIn[0]);
  and and1(a1, s0, ns1, aIn[1]);
  and and2(a2, ns0, s1, aIn[2]);
  and and3(a3, s0, s1, aIn[3]);
  //
  or or1(f, a0, a1, a2, a3);
endmodule
```



**Figure 2**

Verilog code for a 4 to 1 multiplexer

Note that gates declared in this fashion are said to be concurrent. In other words, sequential ordering of gates is not considered; all gates are done in parallel. Table 3, in the Appendix, gives a list of gates supported by Verilog.

- **Continuous Assignment Statement** – In Verilog the **assign** statement is used to assign a value to a net type (**wire** or **tri**) outside of an **always** block. The **assign** statement is implied when you assign a value to a wire during its declaration. Thus **wire w1 = a ^ b;** is the same as **wire w1; assign w1 = a ^ b;** Note that continuous assignment statements are concurrent. In other words, if we write two assignment statements such as:

```
assign w1 = a ^ b;
assign w2 = c | d;
```

Verilog takes the two assignment statements as happening at the same time in parallel and not sequentially as shown. This is very different from most programming languages. The three modules in Figure 3 have the same result. The order of assignment statements is of no importance since all assignment statements are done in parallel.

```

module abs(f,a,b,c); module abc(f,a,b,c); module abc(f,a,b,c);
  input a, b, c;      input a, b, c;      input a, b, c;
  output f;           output f;           output f;
  wire w1;            wire w1 = a & b;     assign f = a&b | c;
  and A1(w1, a, b);    assign f = w1 | c; endmodule
  or(f, w1, c);        endmodule
endmodule

```

**Figure 3**

Three examples of a circuit which implements the logic function  $f = a \cdot b + c$ .

```

module mux4to1Asn(f, s1, s0, aIn);
  output f;
  input s1, s0;
  input [3:0]aIn;
  assign f = ~s1&~s0&~aIn[0] | ~s1&s0&aIn[1] |
             s1&~s0&aIn[2] | s1&s0&aIn[3];
endmodule

```

**Figure 4**

The Verilog code for the 4 to 1 multiplexer of Figure 2 using the assignment operator.

- **Operators** – Operators in Verilog are similar to those in C++. Table 1 gives the most often used operators in summary form and Table 2 gives their precedence order (Tables are in the appendix). The list below shows some examples and notes differences from what would be expected in C++.
  - Bitwise operators** – The bitwise operators work as in C++. The don't care (x) and high impedance (z) states can also be used with these operators. The result is always another variable.
  - Logical operators** – These operators work the same as C++ and return either true(1) or false (0). If an operand has an x or z bit the result is x or z. Note that the logical not operator inverts true or false and that x inverts to x. This is different from the bitwise not operator (~) which returns an inverted bit.
  - Reduction operators** – These operators are not in C++. They operate in a unary fashion on all of the bits in a single variable. For example `y = &x` would make y become the logical AND of all of the bits in x. `xor` and `xnor` can be used to quickly determine parity.
  - Arithmetic operators** – These operators work the same as in C++ except if any bit in an operand is x (a don't care) the result is x.
  - Relational operators** – The relational operators return true (1) or false (0). If any bit is an x (a don't care) these operators return x. If any bit is a z, these operators fail.
  - Equality operators** – The `==` and the `!=` operators are the same as in C++ with the exception that if any operand has an x bit then the result is x. The `===` and `!==` operators compare x to x and z to z and return true (1) or false (0). (`===` and `!==` are for simulation only.)
  - Miscellaneous operators** – The shift operators work the same as in C++. The **concatenation operator** allows you to stick words or bits together to form new words. For example
 

```
c = {a[0], b[7:1]};
```

forms a new 8-bit word consisting of the 0<sup>th</sup> bit of a and the most significant 7 bits of b. The **repetition operator** allows you to concatenate multiple words. For example

$$c = \{3\{a\}\};$$

makes c the same as {a, a, a}

## 4.0 Control Structures – Control constructs, always, if, case, for, and while

- Control Constructs – Verilog's control constructs can be thought of as existing in two categories called **concurrent constructs and sequential constructs**. Concurrent constructs happen in parallel and sequential constructs happen like they are written as in a computer program. The **two concurrent constructs** we will discuss here are **gate instantiation and the continuous assignment statement**. The **sequential constructs** commonly used in Verilog include **the *always* and *initial* blocks, the *if* and *case* structures, and the *for* and *while* loops.**

**Always and Initial blocks** – The *always* and *initial* blocks are similar in structure.

The *Initial* block provides initial values for simulation purposes and does not play a role in circuit synthesis. The *initial* block, then is used in conjunction with a Verilog simulator to establish initial values. For example, initial values may be needed for testing purposes in a simulated environment. For circuit synthesis we use the *Always* block and such a block must contain all sequential constructs. Thus the *if*, *case*, *for* loop, and *while* loop must appear inside an *always* block.

The general syntax for an *Always* block looks like this:

```
Always @(sensitivity_list)
[begin]
    [sequential statements consisting of assignment, if, case,
     while, and for loops. May also include task and
     function calls.
]
[end]
```

The sensitivity list is a list of the variables which, if changed, would produce a different output in the *always* block. The @ sign is referred to as *event control operator*. The sensitivity list is used by Verilog simulators to determine when the *always* block should be executed and updated. The sensitivity list consists of variables separated by the word *or* as in ***always* @(a or b or c)**. Verilog 2001 allows for a comma separated list as in ***always* @(a, b, c)**

Note that the statements in an ***always*** block between ***begin/end*** are executed sequentially just as they are written. For modules that have multiple ***always*** blocks however, all of the ***always*** blocks are executed in parallel.

Variables assigned a value inside and *always* block must be of type ***reg*** or ***integer***. You may not assign type ***wire*** or ***tri*** (nets) a value inside an *always* block. (This would be very confusing since continuous assignments to wires are concurrent operations and the code inside an *always* block is sequential.)

**If block** – The *if* structure works the same as it does in C++ with the exception that Verilog uses the words *begin* and *end* to designate code blocks instead of braces {

and } (braces are used for the concatenation operator.) A typical if structure might look like this:

```

if(t == y)
begin
    z = 12;
end
else
begin
    z = 22;
end

```

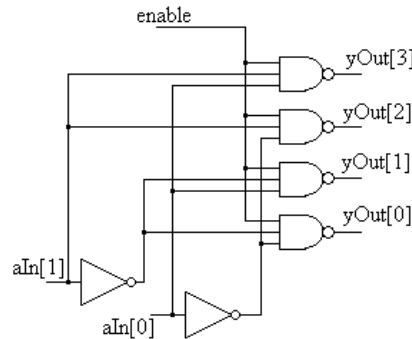
As in C++, the begin and end key words are not needed if the block has only one line. Thus the above example could be written as:

```

if(t == y)
    z = 12;
else
    z = 22;

```

The *if* construct can be used inside an always block to implement combinational or sequential logic. Figure 5 shows a 2 to 4 decoder implemented a sequence of if statements inside an always block.



**Figure 5**

A 2 to 4 decoder circuit.

```

module Decode2To4(aIn, yOut, enable);
    input [1:0]aIn;
    input enable;
    output [3:0]yOut;
    reg [3:0] yOut;
    always@(aIn or enable)
    begin
        if(enable == 1)
        begin
            if(~aIn[1] && ~aIn[0]) yOut = 4'b0111;
            if(~aIn[1] && aIn[0]) yOut = 4'b1011;
            if(aIn[1] && ~aIn[0]) yOut = 4'b1101;
            if(aIn[1] && aIn[0]) yOut = 4'b1110;
        end
        else
            yOut = 4'b1111;
        end
    end
endmodule

```

**Figure 6**

A 2 to 4 decoder in Verilog. An always block and if statements are used for this implementation.

Case structure – the case structure has a slightly different syntax than its counterpart in C++ in that no break statement is necessary and the word *switch* is not used. A typical case structure might look like this:

```

case (t)
  0: y = w[0];
  1: y = w[1];
  ...
  7: y = w[n];
  default: y = x;
endcase

```

In this example each alternative in the case statement takes up only one line.

Multiline blocks can be used with begin and end statements.

There are two other "flavors" of the case statement in Verilog. These are called casex and casez. The original case statement checks the alternatives for an exact match of 0, 1, x, or z. The casex statement treats both x and z as true don't cares. The casez statement treats x as a don't care but demands that the z digit have an exact match.

```

module SevSegCase(aIn, sOut);
  input [3:0]aIn;
  output [6:0]sOut;
  reg [6:0]sOut;
  always @(aIn)
    begin
      case (aIn)
        //          abcdefg
        4'b0000:sOut = 7'b0000001; //0
        4'b0001:sOut = 7'b1001111; //1
        4'b0010:sOut = 7'b0010010; //2
        4'b0011:sOut = 7'b0000110; //3
        4'b0100:sOut = 7'b1001100; //4
        4'b0101:sOut = 7'b0100100; //5
        4'b0110:sOut = 7'b0100000; //6
        4'b0111:sOut = 7'b0001111; //7
        4'b1000:sOut = 7'b0000000; //8
        4'b1001:sOut = 7'b0001100; //9
        4'b1010:sOut = 7'b0001000; //A
        4'b1011:sOut = 7'b1000010; //B
        4'b1100:sOut = 7'b0000111; //C
        4'b1101:sOut = 7'b0000001; //D
        4'b1110:sOut = 7'b0110000; //E
        4'b1111:sOut = 7'b0000110; //F
      endcase
    end
endmodule

```

**Figure 7**

The Verilog code for a 7-segment decoder with active low outputs. This implementation uses a case structure.

For loop – The for loop in Verilog has the same structure that it does in C++. The braces in the C++ structure are replaced by a begin/end block and the ++ operator is not used for the loop counter variable. A typical for loop might look like this:



```

    for(i=0;i<10;i=i+1)
    begin
        s[i] = p[i] ^ q[i];
    end

```

Note that integers are typically used for counter variables.

```

module Decode3To8For(yOut, aIn, enable);
    output [7:0]yOut;
    input [2:0]aIn;
    input enable;
    reg [7:0] yOut;
    integer k;
    always@(aIn or enable)
    begin
        if(enable == 1)
        begin
            for(k=0;k<8;k=k+1)
            begin
                if(aIn == k)
                    yOut[k] = 0;
                else
                    yOut[k] = 1;
            end
        end
    end
endmodule

```

**Figure 8**

A Verilog implementation of a 3 to 8 decoder with active low outputs. This implementation uses a for loop.

**While loop** – The while loop in Verilog is much the same as it is in C++. As in the case of the for loop, the braces in the C++ structure are replaced by a begin/end block. The general form for the Verilog while loop is as follows:

```

    initialize condition
    while(condition)
    begin
        ...
        update condition
    end

```

## 5.0 Sequential Circuits – non blocking assignment, sequential circuit elements, and registers

- **Non blocking assignment statement** – The equals sign is used for the blocking assignment statement as in

```

    a = b;
    c = a;

```

where we assume that a, b, and c are inside an always block and are of type **reg**. The result of these two statements is that both variables a and c take on the value of b. Verilog has another type of assignment statement called the non blocking assignment statement which uses the <= operator instead of the = operator. If a, b, and c are of type **reg** and they are inside an always block then the statements

```

    a <= b;
    c <= a;

```

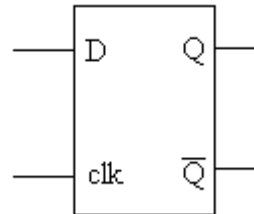
make a take on the value of b in the first statement but c takes on the value that a had at the beginning of the block as if the first statement never occurred. In other words, the non blocking assignment statement evaluates variables by using the value those variables had at the beginning of the block. The non blocking assignment operator allows the two statements to execute concurrently instead of sequentially. In general, in Verilog you should use the non blocking assignment statement for combinational logic and the blocking assignment statement for sequential logic. Likewise, for sanity, you should never mix blocking and non blocking assignment statements in the same always block.

- Sequential circuit elements – The flip-flop is the basic building block of any sequential circuit. Verilog supports many types of flip-flops in library functions and provides a means for a user to define flip-flops for a variety of needs. Consider the classic D flip-flop in which the Q output follows the D input as long as the clock is high. When the clock goes low, the D input value is locked in. (The classic D flip-flop is often referred to as a D latch.) The figure below shows how we can implement the classic D flip-flop in Verilog code.

```

module classicD(D, clk, Q, Qn);
    input D, clk;
    output Q, Qn;
    reg Q, Qn;
    always@(D or clk)
        if(clk)
            begin
                Q <= D;
                Qn <= ~D;
            end
endmodule

```



**Figure 9**

Implementation of a classic D flip-flop in Verilog.

The if statement in this figure incompletely specifies the behavior of the Q output since there is no else clause with the if. This implies memory and Verilog implements this construct with a flip-flop.

The classic D flip-flop is controlled by the level of the clock. That is, when the clock is high the Q output follows the D input. Many other types of flip-flops work on the clock edge which may be either positive going or negative going edges. In Verilog such behavior is modeled using the event controls called **posedge** and **negedge** in the always block sensitivity list. The figure below shows a positive edge triggered D flip-flop with an asynchronous reset and preset. The always declaration in this figure includes the negedge specifier on both resetn and presetn even though these two signals are supposed to be asynchronous. This is because an always block sensitivity list may not contain both edge triggered and level sensitive signals.

```

module DFFAsyncClr(D, clk, resetn, Q, presetn);
  input D, clk, resetn, presetn;
  output Q;
  reg Q;
  always@(posedge clk or negedge resetn or negedge presetn)
    if(!resetn)
      Q <= 0;
    else if(!presetn)
      Q <= 1;
    else
      Q <= D;
endmodule

```

**Figure 10**

An positive edge triggered D flip-flop with asynchronous reset and preset.

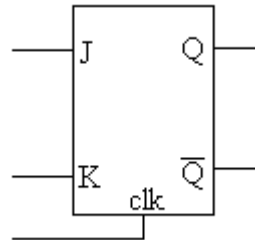
To make a D flip-flop with a synchronous reset and preset we can simply omit the **negedge** event controller from the resetn and presetn signals in the sensitivity list. This causes the system to only look at changing the flip-flop when the clock edge occurs.

A JK master/slave flip flop (JKMSFF) is actually two flip flops in one package. The first is a master flip flop which locks in the data on the rising edge of the clock signal. The second is the slave flip flop which gets data from the master flip flop on the falling edge of the clock. Implementation in Verilog requires two always blocks since one block may not contain both the **posedge** and the **negedge** event control for the same variable (clk in this case). The figure below shows one implementation for a JKMSFF.

```

module jkff(J, K, clk, Q);
  input J, K, clk;
  output Q;
  reg Q;
  reg Qm;
  always @(posedge clk)
    if(J == 1 && K == 0)
      Qm <= 1;
    else if(J == 0 && K == 1)
      Qm <= 0;
    else if(J == 1 && K == 1)
      Qm <= ~Qm;
  //
  always @(negedge clk)
    Q <= Qm;
endmodule

```

**Figure 11**

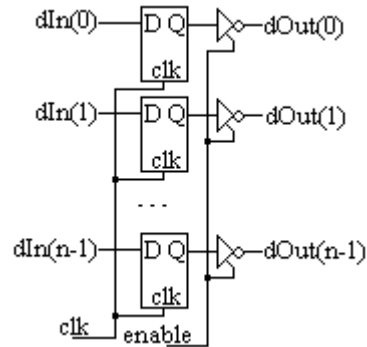
Verilog code for a JK flip-flop. This implementation makes use of two always blocks.

- Registers – a register is by definition a collection of flip-flops or latches that share a common clock and often, a common reset or preset line and a common tri-state output control. The figure below shows an 8-bit register with a tri-state control and its Verilog implementation.

```

//RegnBit
//n-bit parallel in/parallel out
// register with tri-state out.
module RegnBit(dIn, dOut, clk, enable);
    parameter n = 8;
    input [n-1:0] dIn;
    input clk, enable;
    output [n-1:0] dOut;
    reg [n-1:0] dOut;
    reg [n-1:0] state;
    always @(enable)
        begin
            if(enable)
                dOut = state; //data to out
            else
                dOut = n'bz; //tri-state out
            end
        always @(posedge clk)
            state <= dIn;
    endmodule

```

**Figure 12**

A parallel in/parallel out 8-bit register with a tri-state output control.

Shift registers can be implemented efficiently in Verilog using the concatenation operator. If the variable `state` represents an `n`-bit shift register then the two statements:

```

sOut <= state[n-1];
state <= {state[n-2:0], sIn};

```

move the  $n-1^{\text{st}}$  bit to the output and create a new state by concatenating the  $n-1$  bits of the old state with the new input from `sIn`. The figure below shows a complete Verilog implementation.

```

//Shiftn
//n-bit shift register serial in serial out
module Shiftn(clk, sIn, sOut);
    parameter n = 60; //number of stages
    input sIn, clk;
    output sOut;
    reg sOut;
    reg [n-1:0] state;
    always @(posedge clk) //sIn -> [0|1|...|n-1] -> sOut
        begin
            sOut <= state[n-1];
            state <= {state[n-2:0], sIn};
        end
endmodule

```

**Figure 13**

Verilog implementation of an `n`-bit shift register. In this case `n` is set to 60.

Registers can also be put together with some *glue* logic to form counters. In this case, Verilog figures out what logic is needed so that designing a counter is a matter of setting up a register variable that counts on the clock edge. The figure below shows an `n`-bit counter with a mod count parameter. The `modCnt` variable is set to 11 so that

the counter shown counts from 0 to 10 and rolls over. This particular counter does not have parallel load but it does have a `clr` input to initialize it to zero.

```
//Cnt4bit
//n-bit counter with clear
module Cnt4Bit(clk, state, clr);
    parameter n = 4;
    parameter modCnt = 11;
    input clk, clr;
    output [n-1:0]state;
    reg [n-1:0]state;
    always@(posedge clk)
        if(clr)
            state <= 0;
        else
            state <= (state + 1) % modCnt;
endmodule
```

**Figure 14**

A 4-bit counter with a modulus. The `modCnt` variable is set to 11 so that this counter counts 0 to 10 before rolling over.

A sequence counter is one which can count in any sequence. The case structure make implementation of such a counter easy in Verilog. The figure below shows a counter which runs through counts 0, 1, 2, 4, 9, 10, 5, 6, 8, 7, 0, ...

```
//CntSeq.v
//Sequence counter
module CntSeq(clk, reset, state);
    parameter n = 4;
    input clk, reset;
    output [n-1:0]state;
    reg [n-1:0]state;
    //
    always @(posedge clk)
        if(reset)
            state = 0;
        else
            begin
                case (state)
                    4'b0000:state = 4'b0001;    //0 -> 1
                    4'b0001:state = 4'b0010;    //1 -> 2
                    4'b0010:state = 4'b0100;    //2 -> 4
                    4'b0100:state = 4'b1001;    //4 -> 9
                    4'b1001:state = 4'b1010;    //9 -> 10
                    4'b1010:state = 4'b0101;    //10-> 5
                    4'b0101:state = 4'b0110;    //5 -> 6
                    4'b0110:state = 4'b1000;    //6 -> 8
                    4'b1000:state = 4'b0111;    //8 -> 7
                    default:state = 4'b0000;
                endcase
            end
endmodule
```

**Figure 14**

A case statement is used to implement a sequence counter.

## 6.0 Modular Programming – Modularization, parameterized modules, functions, and tasks.

- Modularization – Verilog can be modularized in several ways including functions, multiple parameterized modules, and tasks.

*Multiple parameterized modules* – Parameters may be added to modules in Verilog to make them more general. To add a parameter to a module we use the keyword **parameter** as in the Shiftn example used above which had the following lines:

```
//Shiftn
//n-bit shift register serial in serial out
module Shiftn(clk, sIn, sOut);
    parameter n = 60;           //number of stages
    ...
```

A parameterized module can then be added to a project and instantiated multiple times with different parameters. Figure 15 shows how this can be done.

```
//ShiftMultiple
//This file uses multiple modules for dual shift registers
// of differing lengths
module ShiftMultiple(sIn, sOut, clk);
    input [2:0] sIn;
    input clk;
    output [2:0] sOut;
    Shiftn Shift4 (clk, sIn[0], sOut[0]); //defaults to n = 4
    Shiftn Shift6(clk, sIn[1], sOut[1]);
        defparam Shift6.n = 6;           //resets n to 6
    Shiftn Shift12(clk, sIn[2], sOut[2]);
        defparam Shift12.n = 12;        //resets n to 12
endmodule

//
//Shiftn
//n-bit shift register serial in serial out
module Shiftn(clk, sIn, sOut);
    parameter n = 4;           //number of stages
    input sIn, clk;
    output sOut;
    reg sOut;
    reg [n-1:0] state;
    always @(posedge clk)      // sIn -> [0|1|...|n-1] -> sOut
        begin
            sOut <= state[n-1];
            state <= {state[n-2:0], sIn};
        end
endmodule
```

**Figure 15**

The second module in this program creates an n-bit shift register where n is a parameter. The first module uses the second module to create three shift registers of differing lengths.

*Functions* – Functions in Verilog have some rather severe limitations but are useful for small pieces of code that must be repeated. Function declarations are similar to that of module declarations. In general a function declaration looks like this:

```
function name;
```

```

    input arguments;
    reg variables;
    parameter parameters;
    integer integers;
    Function body
endfunction

```

The function name is like any other variable name and may be a vector. A function can only return the value assigned to its name but this is just an inconvenience and not a serious limitation since values can be concatenated together and packed into a single name. A more serious limitation of functions is that functions must contain only combinatorial logic so that latches and flip-flops may not be coded into functions. Functions may call other functions but they may not call other tasks. Figure 16 shows a Verilog file with an example of a function.

```

module fundecode(aIn, yOut, enable);
    input [1:0] aIn;
    input enable;
    output [3:0] yOut;
    reg [3:0] yOut;
    //
    function [3:0] FindOutput;
        input [1:0] xIn;
        if(~xIn[1] && ~xIn[0]) FindOutput = 4'b0111;
        if(~xIn[1] && xIn[0]) FindOutput = 4'b1011;
        if(xIn[1] && ~xIn[0]) FindOutput = 4'b1101;
        if(xIn[1] && xIn[0]) FindOutput = 4'b1110;
    endfunction
    //
    always@(aIn or enable)
    begin
        if(enable == 1)
        begin
            yOut = FindOutput(aIn);
        end
        else
            yOut = 4'b1111;
        end
    end
endmodule

```

**Figure 16**

This is the 2:4 decoder of Figures 5 and 6 implemented using a function.

All functions must have at least one input but functions may not have any output or inout variables. Functions may call other functions but functions may not call other tasks.

**Tasks** – The task was created for Verilog to alleviate some of the limitations of functions. Tasks may have multiple inputs but unlike functions may be written with zero inputs. Tasks may also have multiple outputs and they may contain events so that they are not limited to combinatorial logic. Tasks may call functions or other tasks in some implementations tasks may be recursive and call themselves. Figure 17 shows a simple implementation of a task.

```

module taskdecode(aIn, yOut, enable);
    input [1:0] aIn;
    input enable;

```

```

output [3:0]yOut;
reg [3:0] yOut;
//
task FindOutput;
  input [1:0]xIn;
  output [3:0] tOut
  if(~xIn[1] && ~xIn[0]) tOut = 4'b0111;
  if(~xIn[1] && xIn[0]) tOut = 4'b1011;
  if(xIn[1] && ~xIn[0]) tOut = 4'b1101;
  if(xIn[1] && xIn[0]) tOut = 4'b1110;
endtask
//
always@(aIn or enable)
  begin
    if(enable == 1)
      begin
        FindOutput(aIn, yOut);
      end
    else
      yOut = 4'b1111;
    end
  end
endmodule

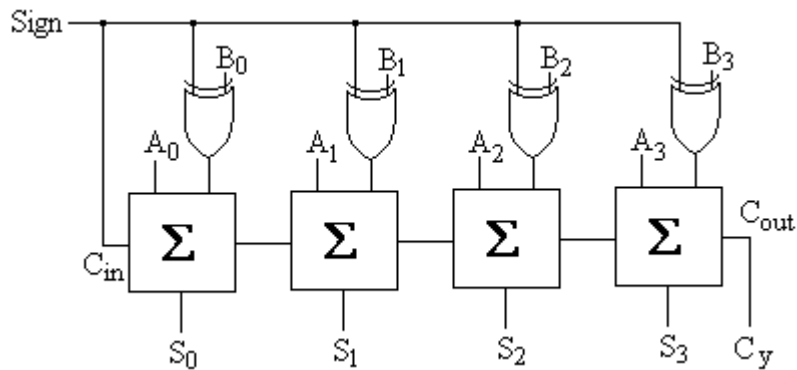
```

**Figure 17**

This is the same 2:4 decoder as shown in Figure 16 except that the function has been replaced by a task.

## 7.0 Examples

Example 1 – A four-bit adder/subtractor.

**Figure E-1.1**

A four-bit adder/subtractor. If Sign = 0 then  $S = A + B$ . If Sign = 1 then  $S = A - B$ .

```

//AddSub.v
//4-bit adder/subtractor
module addsub(aIn, bIn, sum, cOut, sign);
  input sign;
  input [3:0]aIn, bIn;
  output [3:0]sum;
  output cOut;
  reg [3:0]sum;
  reg [3:0]bExOr;

```



```

    reg cOut;
    //
    always @(aIn or bIn or sign)
    begin
        bExOr = bIn ^ {4{sign}};
        {cOut, sum} = aIn + bExOr + sign;
    end
    //
endmodule

```

**Figure E-1.2**

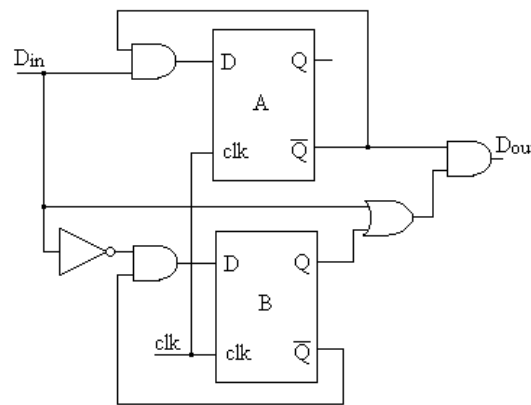
A four-bit adder/subtractor implemented in Verilog

**Example 2 – Manchester data encoder**

```

//Manchester.v
//Does Manchester encoding
module manchester(clk, dIn, dOut);
    input clk;
    input dIn;
    output dOut;
    reg Qa, Qb;
    wire dA, dB, dOut;
    assign dA = dIn & (~Qa);
    assign dB = (~dIn) & (~Qb);
    assign dOut = (~Qa) & (dIn | Qb);
    always@(posedge clk)
    begin
        Qa <= dA;
        Qb <= dB;
    end
endmodule

```

**Figure E2-1**

A Mealy machine for Manchester encoding.

**Example 3 – An n-bit ALU with 16 functions.**

```

module ALUnbit(aIn, bIn, sIn, cIn, bOut, cOut);
    parameter n = 4;
    input [n-1:0] aIn;
    input [n-1:0] bIn;
    input [n-1:0] sIn;
    input cIn;
    output [n-1:0] bOut;
    output cOut;
    reg cOut;
    reg [n-1:0] bOut;
    reg [n:0] tmp;
    always @(aIn or bIn or cIn or sIn)
    begin
        cOut = 1'b0;
        case(sIn)
            0:bOut = ~aIn; //not a
            1:bOut = ~(aIn | bIn); //nor
            2:bOut = ~aIn & bIn; //not a and b

```

```

3:bOut = 0;           //zero
4:bOut = ~(aIn & bIn); //nand
5:bOut = ~bIn;        //not b
6:bOut = aIn ^ bIn;   //ex or
7:bOut = aIn & ~bIn;   //a and not b
8:bOut = ~aIn | bIn;   //not a or b
9:begin               //aIn plus bIn
    tmp = {1'b0,aIn} + {1'b0,bIn};
    bOut = tmp[n-1:0];
    cOut = tmp[n];
end
10:begin               //aIn plus bIn + cIn
    tmp = {1'b0,aIn} + {1'b0,bIn} + {1'b0,cIn};
    bOut = tmp[n-1:0];
    cOut = tmp[n];
end
11:begin               //aIn minus bIn
    tmp = {1'b0,aIn} - {1'b0,bIn};
    bOut = tmp[n-1:0];
    cOut = tmp[n];
end
12:begin               //aIn minus bIn minus cIn
    tmp = {1'b0,aIn} - {1'b0,bIn} - {1'b0,cIn};
    bOut = tmp[n-1:0];
    cOut = tmp[n];
end
13:bOut = ~(aIn ^ bIn); //exclusive NOR
14:bOut = -aIn;          //twos complement
15:bOut = 4'b1;          //all ones
endcase
end
endmodule

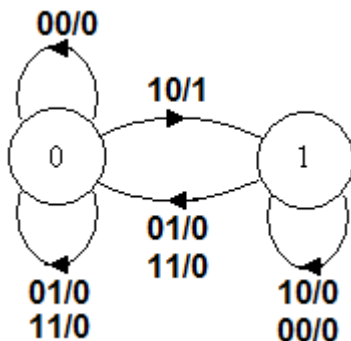
```

**Figure E3-1**

An n-bit ALU with 16 logical and arithmetic functions.

#### Example 4: A Leading ones detector

This circuit detects the first one in a sequence. It has two inputs, x and y. When y is 1 it resets.



```
module SeqLeadingOnes(xIn, yIn, z, clk);
    input clk, xIn, yIn;
    output z;
    reg [1:0] out;
    reg state;
    reg z;
    always @(posedge clk)
        begin
            if(yIn)
                begin
                    state = 0;
                    z = 0;
                end
            else
                case (state)
                    0:
                        if(xIn)
                            begin
                                state = 1;
                                z = 1;
                            end
                        else
                            begin
                                state = 0;
                                z = 0;
                            end
                    1:
                        begin
                            state = 1;
                            z = 0;
                        end
                endcase
            end
        end
endmodule
```

## Appendix A

	Symbol	Comment
Bitwise	~	ones complement unary operator
	&	and
		or
	^	exclusive or
	~&	nand
	~	nor
	~^	exclusive nor
Logical	!	not unary operator also called logical negation
	&&	logical and
		logical or
Reduction	&	and unary operator
	~&	nand unary operator
		or unary operator
	~	Nor unary operator
	^	exclusive or unary operator
	~^	exclusive nor unary operator
Arithmetic	+	add
	-	subtract
	-	two's complement – unary operator
	*	multiply
	/	divide
	%	mod operator
Relational	>	greater than
	<	less than
	>=	greater than or equals
	<=	less than or equals
Equality	===	case equals, compares x and z, simulation only
	!==	case not equals, compares x and z, simulation only
	==	equals, produces x if bits are x or z
	!=	not equals, produces x if bits are x or z
Misc	>>	shift right
	<<	shift left
	? :	(cond)?(statements if true):(statements if false)
	{,}	concatenation
	{m{}}	repetition where m is repetition number

**Table 1**

This table lists the most often used operators in Verilog. See the discussion for examples.

	Operator	Comment
1	! & ~&   ~  ^ ~^ + -	unary highest precedence
2	* / %	arithmetic
3	+ -	
4	<< >>	shifts
5	< <= > >=	relational
6	== != === ~==	equality
7	& ~& ^ ~^	bitwise and reduction
8	~	
9	&&	logical
10		logical lowest precedence

**Table 2**

The precedence of operators in Verilog. Operators on one line in this table are of equal precedence with the one on the left in an expression taking precedence. Parenthesis override all precedence operators and can be used to effectively reorder the precedence.

Gate	Function	Comment
<b>and</b> (f, a, b, ...)	$f = a \cdot b \cdots$	and function
<b>or</b> (f, a, b, ...)	$f = a + b + \cdots$	or function
<b>not</b> (f, a)	$f = \bar{a}$	inverter function
<b>xor</b> (f, a, b, ...)	$f = a \oplus b \oplus \cdots$	exclusive or function
<b>nand</b> (f, a, b, ...)	$f = \overline{a \cdot b \cdots}$	nand function
<b>nor</b> (f, a, b, ...)	$f = \overline{a + b + \cdots}$	nor function
<b>xnor</b> (f, a, b, ...)	$f = \overline{a \oplus b \oplus \cdots}$	exclusive nor function
<b>buf</b> (f, a)	$f = a$	buffer
<b>notif0</b> (f, a, cond)*	$\text{if}(\bar{\text{cond}}) \bar{a} \text{ else } 'bz$	if(cond not true) then $\bar{a}$ else high z
<b>notif1</b> (f, a, cond)*	$\text{if}(\text{cond}) \bar{a} \text{ else } 'bz$	if(cond true) then $\bar{a}$ else high z
<b>bufif0</b> (f, a, cond)*	$\text{if}(\bar{\text{cond}}) a \text{ else } 'bz$	if(cond not true) then $a$ else high z
<b>bufif1</b> (f, a, cond)*	$\text{if}(\text{cond}) a \text{ else } 'bz$	if(cond not true) then $a$ else high z

\*notif and bufif are inverting and noninverting tri-state gates with active low and active high enables.

**Table 3**

Gates available in Verilog and their logical definition.

## References

1. Smith, David R. and Franzon, Paul D., Verilog Styles for Synthesis of Digital Systems, Prentice-Hall, 2000.
2. Mano, M. Morris and Kime, Charles R., Logic and Computer Design Fundamentals, 3<sup>rd</sup> edition, Pearson Prentice-Hall, 2004.
3. Brown, Stephen and Vranesic, Zvonko, Fundamentals of Digital Logic with Verilog Design, McGraw-Hill, 2003.
4. Hyde, Daniel C., "CSCI 320 Computer Architecture Handbook on Verilog HDL", 1997, <http://www.eg.bucknell.edu/~cs320/1995-fall/verilog-manual.html>
5. Ciletti, Michael D., Advanced Digital Design with Verilog HDL, Prentice-Hall, 2003.